55-222

Aspect-Oriented Programming is Quantification and Implicit Invocation

Robert E. Filman
RIACS
NASA Ames Research Center
Moffett Field, CA 94035
rfilman@mail.arc.nasa.gov

Daniel P. Friedman
Computer Science Department
Indiana University
Bloomington, IN 47405
dfried@cs.indiana.edu

Abstract

We propose that the distinguishing characteristic of Aspect-Oriented Programming (AOP) languages is that they allow programming by making quantified programmatic assertions over programs that lack local notation indicating the invocation of these assertions. This suggests that AOP systems can be analyzed with respect to three critical dimensions: the kinds of quantifications allowed, the nature of the interactions that can be asserted, and the mechanism for combining base-level actions with asserted actions. Consequences of this perspective are the recognition that certain systems are not AOP and that some mechanisms are meta-AOP: they are sufficiently expressive to allow straightforwardly programming an AOP system within them.

1. Introduction

This paper is about Aspect-Oriented Programming (AOP) qua programming language. We are interested in determining what makes a language AOP. This work was prompted by a question from Tzilla Elrad, who asked whether event-based, publish and subscribe (EBPS) (for example, [16]) is AOP. After all, in a publish-and-subscribe system, separate concerns can be realized by having concerns subscribe to the events they care about. In thinking about that question, we have come to the belief that two properties, *quantification* and *implicit invocation* (which in the past we have called *obliviousness* [15]) are necessary for AOP. Understanding these relationships clarifies the variety of possible AOP languages and suggests research directions for AOP.

Here we are addressing the structural essence of AOP here, not it's application—somewhat similar to the difference between defining Object-Oriented Programming (OOP) systems in terms of polymorphic methods and inheritance, versus waxing euphoric about objects as the appropriate way to model the world. (Here we take inspiration from Wegner [31], who early in the OOP discussion tried to clarify the dimensions of OOP language design.) Our definition clarifies why some systems that might seem to be AOP are not, and why some systems are stronger than just AOP—their primitives allow straightforward construction of AOP mechanisms at the user level.

2. Local and unitary statements

Programming languages are about writing a structure of *statements* that a compilation or interpretation process will elaborate as a series of primitive directions. (The directions themselves will be a finite text, though their interpretation may be unbounded.) The earliest computer (machine language) programs had a strict correspondence between the program text and the execution pattern. Generally, each programming language statement was both *unitary* and *local*—unitary in that it ended up having effect in precisely *one* place in the elaborated program, and local in that it is almost always proximate to the statements executing around it.

The history (of this part) of programming languages has been about moving away from purely local and unitary languages—about mechanisms that let the programmer separate concepts pragmatically, instead of being tied to saying things just where they happen. The first exceptions to locality were subprograms (*i.e.*, procedures, subroutines, functions.) Subprograms were a great invention, enabling abstracting out some behavior to someplace else. They have many virtues for separating concerns. For example, expertise in, say, Runge-Kutta methods could be embodied in the writer of the Runge-Kutta library. The application programmers would be users of that library. They still had to know something about Runge-Kutta (when it was useful, how to invoke it), and had to locally and *explicitly* call it in their code. The program was still unitary: it exhibited a direct correspondence between one statement in the programming language written, one sequence of machine instructions executed.

Inheritance in object-oriented programming was another important introduction of non-locality. Executing inherited behavior is non-local. How explicit this execution was depended on whether the OO language used send super or mixins.

Send-super systems like Java and Smalltalk allow the programmer to explicitly invoke the behavior of its parent class or classes, without knowing exactly what behavior is being invoked. Adding behavior to classes higher in the class structure allows a limited form of *quantified* program statements—that is, statements that have effect on many *loci* in the underlying code. For example, suppose we wish to introduce a "display" aspect to a program about simulating traffic movement. We will want to make quantified statements like "Whenever something moves (executes its move method), the screen must be updated." Imagine that all things that move are descendants of the class of moveable-object. We can accomplish this with send-super inheritance, if we have a cooperative base-class programmer—one who will consistently follow directions. We make the behavior of the move method in movable-object be the display update and request the programmers of derivative classes invoke send-super at the end of their code. This requires the derived class programmers to know that they have to do something, but relieves them of having

to know what exactly it is that they have to do. We're also restricted with respect to the locus of behavior—we can ask programmers to do the send-super at the start of their code, or at the end, but our directions probably need to be consistent throughout the system.

Requiring cooperation is not good enough. Programmers may fail to be systematically cooperative, the base program may itself be already written, or it may be otherwise out of our control. For true AOP, we want our system to work with programmers who don't have to be thinking about other concerns as they program. The behavior of the separate concern must be *implicitly invoked.* An early example of something close to implicit invocation is *mixin inheritance*, found in MacLisp and Symbolics Lisp [5, 25]. With mixins, the derived-class functionality is determined by assembling the code of the derived class with the advice of its super classes. The aspect programmer can make quantified statements about the code by adding mixins, while the derived class programmer remains (almost) ignorant of these actions. The scope of quantification is controlled by which classes inherit the mixin. That is, we can quantify over the descendants of some superclass, for a given single method. In the screen update example, adding an "after" mixin to movable-object's move accomplishes the automatic update. Except that class inheritance relationships are part of a class's definition, we would have an AOP system (and even this caveat has an exception [12].)

In general,

> *AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs that have no explicit reference to the possibility of additional behavior.*

We want to be able to say, "This code realizes this concern. Execute it whenever these circumstances hold." This breaks completely with local and unitary demands—we can organize our program in the form most appropriate for coding and maintenance. We do not even need the local markings of cooperation. The weaving mechanism of the AOP system can, by itself, take our quantified statements and the base program and produce the primitive directions to be performed.

3. Implicit Invocation

Implicit invocation states that one can't tell that the aspect code will execute by examining the body of the base code. Implicit invocation is desirable because it allows greater separation of concerns in the system creation process—concerns can be separated not only in the structure of the system, but also in the heads of the creators.

Conventional programming languages already have mechanisms for separating concerns, most prominently, subprograms. For AOP to be an interesting technology, it must be because it gets us expressive facilities beyond what conventional languages already provide. Since an AOP system without implicit invocation is insufficiently different from a conventional subprogram call, it would merit only marginal interest.

4. Quantification

AOP is thus the desire to make programming statements of the form

In programs P, *whenever condition* C *arises, perform action* A. (1)

over "conventionally" coded programs *P*. This suggests three major dimensions of concern for the designer and implementer of an AOP system:

- **Quantification:** What kinds of conditions *C* can we specify.
- **Interface:** How do the actions *A* interact with the program *P* and with each other.
- **Weaving:** How will the system arrange to intermix the execution of the actions of program *P* with the actions *A*.

In an AOP system, we make quantified statements about which code is to execute in which circumstances. (Quantification is a less-than-completely-precise term; Appendix A contains an initial attempt to formalize this concept.)

Over what can we quantify? Broadly, we can quantify over the static structure of the system and over its dynamic behavior.

4.1. Static quantification

The static structure is the program as text. Two common views of program text are in terms of the public interfaces of the program (typically methods, but occasionally also public variables) and the complete structure of the program—typically, the parsed-program as abstract syntax tree, (though occasionally the object code[7].)

Black-box AOP systems quantify over the public interface of components like functions and object methods. Examples of black-box systems include Composition-Filters [2], synchronization advice [19], aspect moderators [8] and OIF [14]. A simple implementation mechanism for black-box AOP is to wrap components with the aspect behavior.

Clear-box AOP systems allow quantification over the internal (parsed) structure of components. Examples of such system include AspectJ, which allows (among other things) quantifying over both the calling and accepting calls in subprograms [21, 22], and Hyper/J, whose composi-

tion rules allow quantifying over elements such as the interpretation of variables and methods within modules [26].

A given AOP system will present a quantification language that may be as simple as just allowing aspect decoration of subprogram calls, or complex enough to represent pattern matching on the abstract syntax tree or compiled structure of the program. Understood this way, a clear-box AOP system could allow static quantifications such as "add a print statement to show the new value of any assignment to a variable within the body of a while loop, if the variable occurs in the test of the while loop."

Clear-box and black-box techniques each have advantages and disadvantages. Clear-box techniques require source. They provide access to all the (static) nuances of the program. They can straightforwardly implement "caller-side" aspects (aspects associated with the calling environment of a subprogram invocation). Black-box techniques are typically easier to implement (in environments like Lisp, where calls are ordinarily routed through a modifiable function symbol, they can be downright trivial) and can be used on components where the source code is lacking.

Because black-box techniques can't quantify over anything besides a program's interface, clear-box techniques are especially useful for debugging. For example, a clear-box system could implement a concern like a statement-execution counting profiler, or writing to a log file on every update of a variable whose name starts with "log." However, black-box techniques are more likely to produce reusable and maintainable aspects—an aspect tied to the code of a module can easily slip into dependence on the coding tricks of that module. Interfaces imply contracts.

Clear-box techniques are more difficult to implement, as they usually imply developing a major fraction of a compiler. A typical clear-box implementation of structural quantification needs to obtain a parsed version of the underlying program, run transformation rules realizing the quantified aspects over that abstract syntax tree, and output the resulting tree back in the source language for processing by the conventional language compiler. That can be a lot of work.

4.2. Dynamic quantification

Dynamic quantification is tying the aspect behavior to something happening at run-time. Examples of such occasions include

- The raising of an exception.
- The call of a subprogram X within the temporal scope of a call of Y. (The call of X within the context of Y problem is an instance of the "jumping aspect" problem [2].)
- The size of the call stack exceeding some value.
- Some pattern of more primitive events in the history of the program being matched. For example, after the "try password" routine has failed five times, with no intervening successes.

The abstractions most programming languages present about the structure and execution of a program are only a subset of the possible available abstractions: Scheme allows a programmer to capture the "current context" and reinvoke the current behavior. C programmers glibly rummage around on the stack, content in the knowledge that the pattern of procedure calls is straightforwardly recognizable so long as the machine and compiler remain constant. 3-Lisp and similar reflective systems allow the programmer access to the interpreter's state [11]. Elephant allows reference to previous variable values [24]. The ability to program with respect to such properties is an ingredient of programming language design. Even if such elements are absent in the underlying language, an aspect language may still allow quantification over them.

5. Implementation issues

Assertion (1) suggests a design space for AOP languages. It implies choices in each of the three dimensions: quantification, interface, and weaving.

- **Quantification**. Quantification incorporates the notions of defining the "join points" of the code along with the language and mechanisms for selecting when a particular join point deserves a particular aspect. Examples of possible join points include subprogram calls, variable references and statements. As mentioned earlier, one can quantify over the static structure of the program or over its dynamic behavior. Examples of the predicates one can use to describe a static quantification include by package, by the inheritance structure of the program, by the structure of call arguments, by the lexical structure of program element names, and by the nested structure of program elements. (Masterscope is an early example of a quantification language for programs that has a rich language for describing points in program structures [28].) Examples of dynamic quantification scope include the dynamic nesting structure of calls, and the occurrence of particular events (**e.g.,** "after x is assigned 3, while y is greater than 7"). There have also been suggestions that the program of the base code could be provided mechanisms to prevent aspect interactions, and that the system check for incompatible aspect applications.
- **Interface.** Interface includes the structure of the "aspect code," the interactions among aspects, and the relationships and information sharing among the aspects and base code. Issues of interface include what context of the underlying program is available to an aspect, how aspects communicate among themselves and with the underlying program, ordering of aspects at the same locus, and aspect parameterizations
- **Weaving.** Weaving expresses how the system arranges to intertwine the execution of the base code and aspects. Key elements include the actual weaving mechanism (for example, compile-time weaving, altering the interpretation process, or meta- or reflective mechanisms) and the ability to dynamically change quantifications in a running system.

6. Aspect-Oriented Languages

To return to Elrad's question, what qualifies as an aspect-oriented language? Let us consider some possibilities:

- **Rule-based systems.** Roughly, rule-based systems like OPS-5 [4] or, to a lesser extent, Prolog are programming with purely dynamically quantified statements. Each rule says, "Whenever the condition is true, do the corresponding action." (We are ignoring the tendency of rule-based systems to execute only one matching rule.) If we all programmed with rules, we wouldn't have AOP discussions. We would just talk about how rules that expressed concerns *X*, *Y*, and *Z* could be added to the original system, with some mention of the tricks involved in getting those rules to run in the right order and to communicate with each other. The base idea that other things could be going on besides the main flow of control wouldn't be the least bit strange. (One recent paper proposed doing AOP with AI style inference [23].)

  But by and large, people don't program with rule-based systems. This is because rule-based systems are notoriously difficult to program. They've destroyed the fundamental sequentially of almost everything. The sequential, local, unitary style is really very good for expressing most things. The cleverness of classical AOP is augmenting conventional sequentially with quantification, rather than supplanting it wholesale.
- **Event-based, publish and subscribe**. In EBPS systems, the subscription mechanism is precisely a quantification mechanism. ("Let me know whenever you see something like ..."). The question is then, is EBPS implicitly invoked? If the application's programming style is to use events as the interface among components or if the underlying system automatically generates interesting events, then EBPS can be used as a black-box AOP mechanism. On the other hand, if we expect the programmer to scatter event generation for our purposes throughout otherwise conventional programs, then EBPS is not implicit and hence, not AOP.
- **Frameworks.** Framework systems [9] provide a high-level organization (a main flow of control) into which the application programmer plugs in behavior at particular points. In some sense, a framework comes with a particularly defined set of concerns and allows plugging in (and separately specifying) just those concerns. Often frameworks will have default behaviors for these concerns. While frameworks provide a way of separating certain concerns, their restriction to a set of predefined concerns keeps the framework mechanism from rising to the level of a *language* for separating concerns.
- **Intentional Programming and Meta-programming**. Intentional programming (IP) [1] and meta-programming (MP) [20] provide the ability to direct the execution order in arbitrarily defined computational patterns. They can be seen as environments for writing transformation compilers (that is, a mechanism for implementing clear-box AOP), rather than as self-contained realizations of the AOP idea.
- **Generative Programming.** Similarly, generative programming [10] works by transforming higher-level representations of programs into lower-level ones (that is, by compiling high-level specifications.) By incorporating aspects into the transformation rules, one can achieve AOP in a generative programming environment.

7. Closing Remarks

We have identified AOP with the ability to assert quantified statements over programs without explicit reference to these statements. This implies

- **AOP is not about OOP**. OOP is a popular programming language technology. Most implementations of new language ideas are done in the context of OOP. The class hierarchy of OO systems is a convenient structure over which to quantify. However, "quantification" and "implicit invocation" are independent of OO. Therefore, it is perfectly

reasonable to develop AOP for a functional or imperative languages. Work illustrating this point include Coady et. al's AspectC for C [6] and Wand's ADJ/PROC for a first-order functional language [29, 30].

- **AOP is not useful for singletons.** If one has an orthogonal concern that is about exactly one place in the original code, and that orthogonal concern will not propagate to other loci as the system evolves, it is probably a bad idea to use AOP for that concern. Just write a call to the aspect procedure into that place, or permute the source code in whatever way necessary to achieve the aspect. The cost to the software maintenance and evolution process by the existence of an additional aspect probably exceeds the benefit of using that aspect in a single place.
- **Better AOP systems are more implicit.** They minimize the degree to which programmers (particularly the programmers of the primary functionality) have to change their behavior to realize the benefits of AOP. It's a really nice bumper sticker to be able to say, "Just program like always, and we'll be able to add the aspects later." (And change policies later, and we'll painlessly transform the code for that, too.) Realizing that bumper sticker is a challenge to the developers of AOP.

Acknowledgements

Earlier versions of these ideas have been expressed at the AOP workshops in Minneapolis [15] and Budapest [13]. Our thanks to Gregor Kiczales, Diana Lee and Tarang Patel for comments on the drafts of this and its predecessors.

Appendix A: Formalizing the Notion of Quantification

What is quantification? Informally, we have the idea that some programmer has written some code, m, composed, say, of actions $<x, y, z>$. If one thinks about this in terms of implementing the system entirely by an interpreter [17], the action of defining m has stored the association between m and $<x, y, z>$. To execute m, the interpreter retrieves the value for m, getting $<x, y, z>$. This interpreter is exhibiting *preservation of assignment*: If something is stored in a location, and nothing comes along to change that assignment, then retreiving from that location will get back the original value.

Imagine a join point at y, where aspect a is to execute before y. To make this happen, the programmer has likely made an assertion of the following form: "In situations that match pointcut/predicate $\mathcal{P}$, 'before' aspect a applies. If $\mathcal{P}(y)$ is true, when our interpreter goes for the value of m, it gets back $<x, a, y, z>$. But there has not been an assertion of that value—that is, no user statement said, "define m to be $<x, a, y, z>$" Rather, the system has combined the base assertion: "m is $<x, y, z>$" with the quantified statement about a pointcut $\forall\ k.\mathcal{P}(k) \rightarrow a$ **before** k. We thus understand that the interpreter of an Aspect-Oriented Programming system is characterized by not preserving assignment. Rather, such an interpreter has done some theorem proving—it has instantiated a universally quantified formula about when an aspect applies with a particular piece of

code, and implicitly redefined a method definition with the result. Key elements of this are that the base assertion and the aspect assertion can occur in either order, and that the aspect assertion can modify many otherwise unrelated base assertions.

This is not an assertion about what mechanism the AOP system used to accomplish this result—it could have arisen from a compiler [21, 22, 26], by wrapping [2, 8, 14], by load-time transformation [7], by a meta-interpreter [27], or by any number of mechanisms that haven't been invented yet. It's an argument that separately specified behaviors have come to be executed together, and that some of these separate specifications were about groups of "join points." Nor does the use of "theorem proving" imply that the process must be somehow intractable—the theorem proving in question is as complex as the predicates the language designer provides and is over the "closed world" of the specific program in question, and is therefore likely to be quite computationally tractable.

References

1. Aitken, W., Dickens, B., Kwiatkowski, P., de Moor, O., Richter, D., and Simonyi, C., "Transformation in intentional programming," in Devanbu, P. and Poulin, J. (Eds.) Proc. 5th Intl Conf. on Software Reuse, Victoria, Canada, IEEE Computer Society Press, June 1998, pp 114–123. http://www.research.microsoft.com/ip/overview/TrafoInIP.pdf
2. Bergmans, L., and Aksit, M. Composing crosscutting concerns using composition filters. *Comm. ACM vol. 44*, no 10, 2001, pp. 51–57.
3. Brichau, J., De Meuter, W., and De Volder, K. Jumping aspects. Workshop on Aspects and Dimensions of Concerns, ECOOP 2000, Cannes, France, June 2000.
4. Brownston, L., Farrell, R., Kant, E. and Martin, N. *Programming Expert Systems in OPS5.* Reading, Massachusetts: Addison-Wesley, 1985.
5. Cannon, H. Flavors: A non-hierarchical approach to object-oriented programming. Symbolics Inc. (1982).
6. Coady, Y., Kiczales, G., Feeley, M. and Smolyn, G. Using AspectC to improve the modularity of path-specific customization in operating system code. To appear in *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.
7. Cohen, G. Recombining concerns: Experience with transformation. In First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (OOPSLA '99). http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99/position-papers/ws23-cohen.pdf

8. Constantinides, C. A., and Elrad, T. Composing concerns with a framework approach. International Workshop on Distributed Dynamic Multiservice Architectures, 21st International Conference on Distributed Computing Systems (ICDCS-21). Phoenix, Arizona, April, 2001, Vol. 2, pp. 133–140.

9. Cotter, S. and Potel, M. *Inside Taligent Technology.* Reading, Massachusetts: Addison-Wesley, 1995.

10. Czarnecki, K., and Eisenecker, U.W. *Generative Programming: Methods, Tools, and Applications.* Boston: Addison Wesley, 2000.

11. des Rivieres, J. and Smith, B. C. The implementation of procedurally reflective languages. Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pp. 331–347, Austin, Texas, August 1984.

12. Fikes, R. E., and Kehler, T. The role of frame-based representation in reasoning, *Comm. ACM, 28* (1985) pp. 904–920.

13. Filman, R. E. What Is Aspect-oriented programming, revisited, Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming, Budapest, June 2001. http://trese.cs.utwente.nl/Workshops/ecoop01asoc/papers/Filman.pdf

14. Filman, R. E., Barrett, S., Lee, D. D., and Linden, T. Inserting ilities by controlling communications. *Comm. ACM,* in press.

15. Filman, R. E., and Friedman, D. P. Aspect-oriented programming is quantification and implicit invocation . Workshop on Advanced Separation of Concerns, OOPSLA 2000, Oct. 2000, Minneapolis. http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/filman.pdf

16. Filman, R. E. and Lee, D. D. Managing distributed systems with smart subscriptions. Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), Las Vegas, June 2000, pp. 853–860.

17. Friedman, D. P., Wand, M., and Haynes, C. T. *Essentials of Programming Languages (2nd Edition).* Cambridge, MA: MIT Press, 2001.

18. Garlan, D., and Notkin, D. Formalizing Design Spaces: Implicit invocation mechanisms. VDM '91 Formal Software Development Methods, Lecture Notes in Computer Science 551, Springer-Verlag, Berlin, 1991, pp. 31–44. We are using "implicit invocation" in a difference sense than Garlan and Notkin. Garlan and Notkin were defining a style without static references in the source text (much like method invocation in object-oriented systems), while our implicit invocation has no local reference to signal the additional actions.

19. Holmes D., Noble J. and Potter J., Towards reusable synchronisation for object-oriented languages. Aspect-Oriented Programming Workshop, ECOOP'98, July 21, 1998. http://www.mri.mq.edu.au/~dholmes/research/aop-workshop-ecoop98.html

20. Kiczales, G., des Rivieres, J., and Bobrow, D. *The Art of the Metaobject Protocol.* Cambridge, MA: MIT Press, 1991.

21. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of AspectJ, *Proceedings ECOOP 2001,* J. L. Knudsen (Ed.) Berlin: Springer-Verlag LNCS 2072, pp. 327–353.

22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. Getting started with AspectJ. *Comm. ACM vol. 44*, no 10, 2001, pp. 59–65.

23. Laddaga, R., Robertson, P., and Shrobe, H. Aspects of the real-world. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/23-robertson.pdf.

24. McCarthy, J. Elephant. http://www-formal.stanford.edu/jmc/elephant.html.

25. Moon, D. A. Object-oriented programming with flavors. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86) *ACM SIGPLAN Notices, vol. 21*, no. 11, 1986, pp. 1–8.

26. Ossher, H. and Tarr, P. The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. *Comm. ACM vol. 44*, no 10, 2001, pp. 43–50.

27. Sullivan, G. T. Aspect-oriented programming using reflection and meta-object protocols. *Comm. ACM vol. 44*, no 10, 2001, pp. 95–97.

28. Teitelman, W. and Masinter, L. The Interlisp programming environment, *Computer vol. 14*, no. 4, 1981, pp. 25–34.

29. Wand, M. A semantics for advice and dynamic join points in aspect-oriented programming. In *Proceedings of SAIG '01*, Lecture Notes in Computer Science 2196, September 2001. Springer-Verlag, Berlin, 2001, pp. 45.

30. Wand, M., Kiczales, G., and Dutchyn, C. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming, July 2001, ftp://ftp.ccs.neu.edu/pub/people/wand/papers/w-k-d-01.ps

31. Wegner, P. Dimensions of object-based language design, *Proceedings OOPSLA '87, ACM SIGPLAN Notices, 22* (12) December 1987, pp. 168–182.